Lessons from Visualizing Software Architecture Structure Conformance at Thermo Fisher Scientific

Filip Zamfirov[1 [0009-0009-3654-4935]], Andrei Radulescu[2], Jacob Krüger[1 [0000-0002-0283-248X]], and Michel R.V. Chaudron[1 [0000-0001-7517-6666]]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
`f.zamfirov | j.kruger | m.r.v.chaudron@tue.nl`
[2] Thermo Fisher Scientific, Eindhoven, The Netherlands,
`andrei.radulescu@thermofisher.com`

Abstract. Modern software-intensive systems are large and complex. Therefore, well-defined software architectures are required to manage such systems. However, implementing and maintaining a software architecture can be challenging in practice. For instance, architecture erosion can cause the implemented architecture to deviate from the intended one. Industrial practitioners often lack effective tools to check that an evolving software system continues to conform to its intended architecture. In this paper, we share the lessons we learned from adopting, extending, and applying a state-of-the-art architecture visualization tool for conformance checking the structure and dependencies between intended architecture (i.e. subsystems and their relations) and implemented architecture (i.e. implemented subsytems and their dependencies in the codebase) on a large industrial software project. Specifically, we collaborated with Thermo Fisher Scientific and used a graph-based visualization tool on one of the company's systems. Using the tool, we can create a hierarchical view of layered software architectures including subsystem and component dependencies. During demonstration sessions, we presented the visualizations to 14 experts at Thermo Fisher Scientific who are involved with different subsystems to elicit their feedback. Using our tool, the experts found it easier to focus on relevant areas of the system and to detect architecture violations and anomalies. The experts expressed great enthusiasm for using the tool on their own. Our insights suggest that software architecture visualization tools can aid software architects in maintaining the conformance between intended and actual software architectures. However, applying existing tools faces challenges regarding scalability, usability, and the integration into company workflows. Our lessons highlight opportunities for future research and improvements in (open-source) software architecture tools.

Keywords: Software Architecture Conformance · Software Architecture Erosion · Dependency Analysis · Reference Architecture · Experience Report

1 Introduction

Software-intensive systems have become a key enabler of innovative high-tech systems in all areas of society. The growing size and complexity of software in such systems make it increasingly difficult to develop and maintain them [8]. To deal with growing software, defining and implementing a suitable architecture is key [21]. A software architecture defines the organization of a software system from various perspectives, including its structure (how the system is decomposed into subsystems) and behavior (how these subsystems interact) [9]. Consequently, software architectures are key instruments for managing the complexity of a software system.

One practical problem of software architectures is their erosion [17,21]. Architecture erosion is a multifaceted phenomenon that occurs when the implemented architecture violates the intended one, when the internal structure of a system is compromised by a faulty design, or when the design of the system becomes increasingly difficult to modify. Architecture erosion can be analyzed from different perspectives (e.g., quality, structure, evolution), but is most commonly described in terms of violations [17].

In this perspective, architecture erosion represents the violation of design principles, architectural constraints, or architectural rules; all of which lead to poor maintainability of a system [21]. Previous research has proposed different solutions for detecting architectural violations and applied these solutions to open-source software [17]. In contrast, little research has been conducted in industrial settings, which limits the generalizability of the findings [17,25,26]. Furthermore, Wan et al. [26] performed an interview survey with industry practitioners in which they found that they are challenged by a lack of feasible tool support for monitoring architecture conformance and violations.

In this paper, we explore the benefits and challenges of adopting tooling to support software architecture structural conformance checking in practice and to provide industrial insights. To this end, we share an experience report with lessons learned on visualizing and analyzing the architecture conformance of an industrial software system. More precisely, we focus on a specific aspect of software architecture conformance, namely the conformance of the structural aspects of an architecture: The dependencies between architectural subsystems and components and their implemented structural dependencies in the codebase. To do so, we extended and tailored the ARViSAN tool that was developed by Kakkenberg et al. [10]. We used ARViSAN to create interactive views and analyses of industrial software-intensive systems developed at Thermo Fisher Scientific. Through our work, we contribute:

1. We report on our ongoing development of the ARViSAN tool to meet the needs for checking structural architecture conformance.
2. We discuss the challenges, benefits, and insights of applying our extended ARViSAN at Thermo Fisher Scientific, including feedback from 14 software engineering professionals.
3. We share an anonymized dataset detailing the structure and dependencies of an industrial software-intensive system [28].

4. We publish our extended version of ARViSAN, including its parser [27], frontend [12] and backend [11] that we developed.

Through these contributions, we aim to foster future research on checking architecture conformance in practice. Specifically, researchers and practitioners can reuse our tooling and use the shared dataset as ground truth representing a real-world industrial system.

2 Context and Related Work

In this section, we introduce Thermo Fisher Scientific and discuss related work.

2.1 Thermo Fisher Scientific

Thermo Fisher Scientific is a global supplier of analytical instruments and services for laboratories, pharmaceuticals, and biotechnology. In 2024, Thermo Fisher Scientific had an annual revenue of $ 42.9 billion, and employed around 125,000 people. At Eindhoven, Thermo Fisher Scientific develops its Transmission Electron Microscopy (TEM) technology for sample analyses at ultra-high resolutions, reaching sub-Angstrom levels.

TEM microscopes are software-intensive systems composed of various complex instruments. Such instruments include, for instance, detector, sample-handling, vacuum, electromagnetic, and electrostatic devices, which work together through a coordinated software architecture. Each type of device is managed by its own software subsystem, which captures the specific functionalities of the device. This architecture creates a complex multi-level and multi-technology software ecosystem, consisting of several architectural layers. The most basic layer represents components, each of which can contain one or more software projects that contribute to an executable or a library. Related components are grouped into (higher layer) subsystems, each encapsulating the functionalities of a device in the TEM. Finally, subsystems are clustered into subsystem groups providing a family of related functionality.

The software architects at Thermo Fisher Scientific have developed architecture documentation and a reference architecture to ensure the quality of the software architecture on all levels. Both of these resources define architectural rules that shall be applied to subsystems and their dependencies. However, due to the complexity, size, and the speed at which the overall system evolves, architects lack a straightforward way to obtain an overview of the current architecture. Often, they have to piece together information from scattered or outdated sources. Also, it can be challenging to assess and ensure conformance between the reference architecture and the latest implementation. For instance, as for any larger system, the question can arise whether the actual architecture must be aligned to the reference, or vice versa. Lastly, the limited tool support for monitoring architecture and implementation conformance increases the manual effort required to gather the necessary information. To address such challenges, we have initiated a research-industry collaboration in which we started to explore how to apply state-of-the-art solutions for checking structural software-architecture conformance.

2.2 Related Work

Industrial Studies of Architecture Conformance and Violations. Sas et al. [25] conducted a study on the evolution of architecture smells at ASML, in which they extended ARCAN [4] to support the proprietary C/C++ used by the company. Using the extended ARCAN, Sas et al. detected architecture smells across releases for nine projects from one of the company's software systems. They also interviewed developers and architects to learn about maintenance issues the interviewees were experiencing with the projects. Their findings show that architecture smells can spread over time to more artifacts, and that some of the artifacts may suffer from multiple smells. The interviewees linked the affected components with frequent changes, the presence of severe bugs, and general maintenance issues.

Martini et al. [19] studied the impact and refactoring costs of architecture smells to aid practitioners prioritize architectural technical debt. They used ARCAN on multiple industrial systems written in Java to identify three architecture smells. The authors then conducted a survey with the developers to investigate how they prioritize architectural technical debt and how they perceive its impact. Martini et al. report that the practitioners found it useful to automatically detect architecture smells, even low-priority ones. The study emphasizes the impact and refactoring costs of code smells, with the hub-like dependency being the easiest to detect and refactor.

Fontana et al. [3] conducted a study akin to that of Martini et al., utilizing ARCAN on three systems from a software consulting company. Through surveys, they gathered developer feedback on architecture smells identified by the tool. The respondents acknowledged the impact of smells on maintainability, but were unfamiliar with the definitions of many of the eight smells examined. Similar to Martini et al., the findings indicate that developers consider hub-like dependencies as primary candidates for refactoring. Additionally, some of the smells were deemed relevant only within a layered architecture.

Mo et al. [20] explored architecture maintainability in a study of nine industrial systems. They considered two metrics and architectural "hotspots" linked to high maintenance costs. Mo et al., analyzed the systems using their DV8 tool suite and conducted interviews with practitioners. They report that DV8 was key for pinpointing hotspots for refactoring, and DV8 was adopted in the respective company to quantify maintenance costs.

Groot at al. [7] report an interview survey with 17 software developers at ASML. They focused on unintended software dependencies, which represent violations of architecture rules, in multi-lingual software systems. They contribute a catalog of eight unintended dependencies and discuss their overlap with architecture smells and other signs of architecture erosion. An important insight is that resolving unintended dependencies is often delayed, even though they challenge developers' comprehension, because the system still works, other tasks are perceived as more important, and due to the costs involved.

Such studies motivate the need for more advanced tooling to facilitate architecture conformance checking. Unlike such studies, which rely on code parsing to derive the implemented architecture or interviews, we start from a documented reference architecture. We use this specification to construct a detailed view of a system's intended architecture, providing a baseline against which the implemented architecture can be compared. Then, we extract dependencies from the implemented system to visualize violations of the reference architecture. So, besides contributing complementary experiences to the previous work, we also propose a different way of checking architecture conformance: combining the perspectives of intended reference and actually implemented architecture.

Tools for Visualizing and Checking Software Architecture. Previous works have proposed various tools for analyzing and checking implemented software architectures. Azadi et al. [1] and Li et al. [17] provide detailed overviews of such tools, which we summarize. Some tools, such as JArchitect [17], ARCAN [4], DV8 [20], NDepend [16], and Lattix [16], offer graph-based or matrix-based visualizations of an implemented architecture. Other tools, for example, Understand [17], the Renaissance approach [2], and Axivion [17], aim to facilitate dependency analyses. Some tools have been extended to visualize, particularly the evolution of architecture smells [5,6]. All these tools provide powerful capabilities, but they typically derive architectural information by parsing source code. This limits their applicability to specific languages and technologies, with extensions requiring to design new parsers. Moreover, we are not aware of any tool that takes a reference architecture into account. In contrast, our tooling is designed not to require language-specific code parsing and uses a reference architecture. So, we complement existing tools, particularly to help architects reflect on the mapping between reference and implemented architecture.

3 Visualizing the TEM Architecture

Next, we describe the TEM system and explain how we adjusted ARViSAN to visualize its architecture.

3.1 TEM Components and Dependencies

TEM is a complex embedded software system comprising 31 subsystems (cf. Figure 1), each of which provides self-contained functionality. Moreover, each subsystem can be further decomposed into multiple individually built components. To facilitate the build process, each component must contain a metadata file specifying its dependencies to other components.

Originally, a simple solution existed to visualize the TEM software architecture. Specifically, a full component dependency graph was constructed during the system's build process to collect information about its dependencies. As a component was built, its dependencies were gathered from the metadata file. A

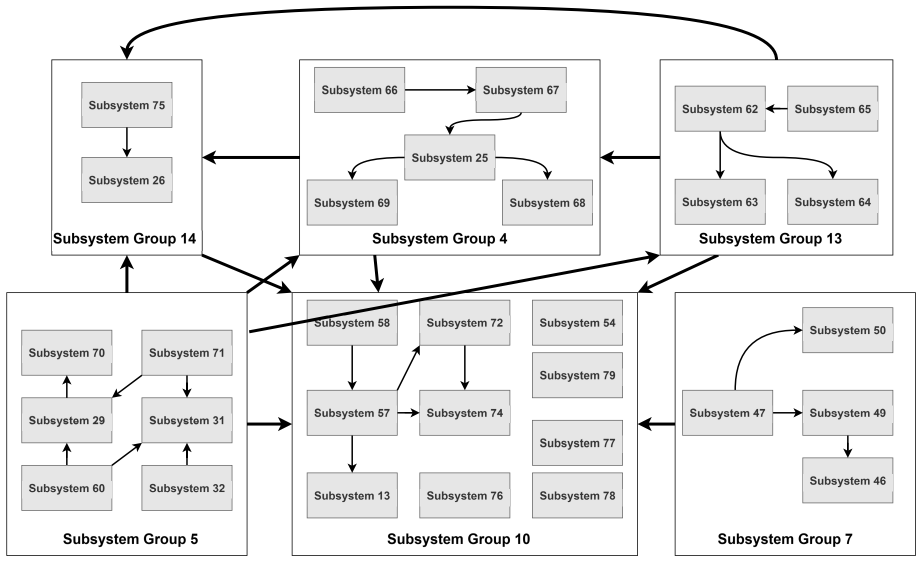

Fig. 1. Anonymized dependency graph depicting the expected high-level dependencies defined in the TEM reference architecture.

script then generated a Graphviz[3] dependency graph. Due to the complexity and size of the TEM software, the graph contained over 200 components and more than 6,000 dependencies, making it very difficult to interpret.

3.2 TEM Reference Architecture

For TEM, Thermo Fisher Scientific architects have defined a reference architecture using the 4+1 architectural view model [13]. To achieve full architecture conformance, all the different views would have to be considered. However, in the context of this work, we focus on the structural aspects of the software system. Therefore, we consider the logical view, which describes the functionality that a system provides to end-users. In this logical view, the architects have specified dependencies on two levels to simplify how dependencies are defined and maintained:

1. dependencies between subsystems within a subsystem group and
2. dependencies between subsystem groups.

In Figure 1, we display an anonymized graph of the intended dependencies in the TEM system.

Using the list of components gathered during the build process and the reference architecture, we created a specification containing all the components

[3] https://graphviz.org/

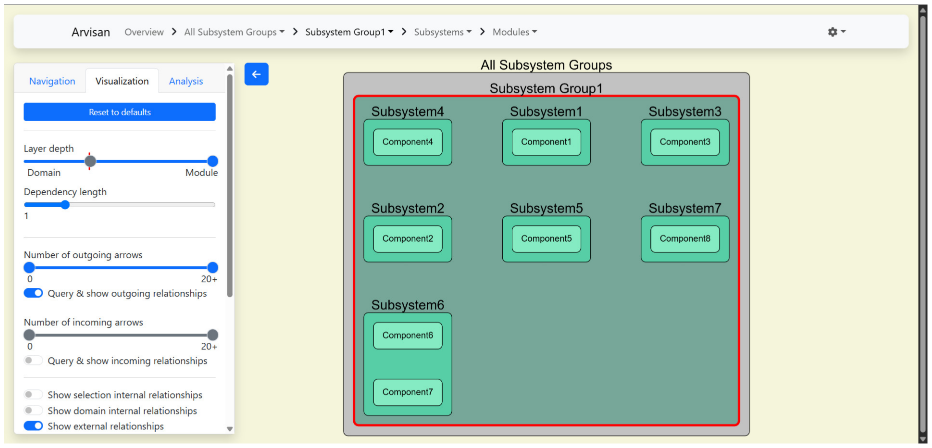

Fig. 2. An anonymized TEM subsystem group (dark green) as displayed in our adapted ARViSAN. The subsystem group includes seven subsystems (green) with one subsystem comprising two components, while the others have one component each (light green).

mapped to a subsystem and a subsystem group. As subsystem interfaces are contained in individual components, the specification also contained metadata about the presence and types of interfaces in a component. We share the anonymized version of this specification to allow other researchers to work with it.[3]

3.3 Visualizing the Architecture

To create an overview of the TEM architecture, we adapted ARViSAN [10]. ARViSAN allows users to interactively explore a graph-based visualization of a target system and its dependencies. We chose to build on ARViSAN because it provides a hierarchical view in which users can explore the system architecture and dependencies at different levels of abstraction. So, ARVISAN was already capable of representing the multiple levels of components, subsystems, and subsystem groups that are relevant for TEM.

Our adapted ARViSAN uses a simple input format: Two `.csv` files of which one defines the logical entities, such as subsystems, and the other defines the implemented dependencies after validation against the reference architecture. These entities and dependencies are the respective nodes and edges that ARViSAN shall visualize. Nodes can be flat or hierarchical, and we distinguish three types of nodes: subsystem groups, subsystems, and components.

Additionally, the input files contain containment relations between the (hierarchical) nodes. This containment information allows to use a slider in ARViSAN to simplify or expand the graph to show higher-level nodes (e.g., subsystem groups) or lower-level nodes (e.g., components). In Figure 2, we exemplify how the containment of components and subsystems in a subsystem group is rendered in our adapted ARViSAN. The processing of the implemented and the reference architecture dependencies is described in our parser repository and can be replicated using the anonymized data that we share.

3.4 Visualizing Architecture Violations

The reference architecture provided to ARViSAN defines the original baseline architecture of a system with its intended dependencies. To enable engineers to identify violations more easily, we integrated a parser into ARViSAN that compares the dependencies between the two architectures and marks differences. Using optional interface information from the specification, we identify dependencies between subsystems in which a component relies directly on the implementation of another subsystem's functionality, rather than its interface. In other words, we check for dependencies that bypass defined interfaces and mark them as potentially degrading.

We extended ARViSAN to display three types of dependencies through colored arrows in the user interface (cf. Figure 3):

Black for conforming dependencies.
Red for non-conforming dependencies (i.e., violations).
Orange for potentially degrading dependencies.

Furthermore, we implemented three coloring modes that change the color of nodes to show component information. Such information includes, for example, the presence of interfaces, their deployment platforms, and their instability. As these coloring modes require company data, not all of them are available in the version of ARViSAN that we share. Finally, we modified ARViSAN to allow nodes to be enriched with custom information like metrics (e.g., instability [18]). Such custom information can be inserted into ARViSAN's input and can be used to better support different architecture analyses.

Through these visualizations and additional interactivity features, we support users of ARViSAN in identifying where the reference- and actual architecture do not conform. For example, a user can focus on an individual node (e.g., a subsystem) by right-clicking it. This renders only those dependencies that are relevant to this node. Furthermore, users can lift the visualized depth to a higher level (e.g., from component to subsystem level) to simplify the view. To illustrate this, we display a circular dependency between two subsystems caused by an unintended dependency in Figure 3 and in Figure 4. In Figure 3, we show the dependencies on the component level, where the circular dependency may be overlooked due to the number of components and dependencies. Considering Figure 4, the dependencies are lifted to the subsystem level, and the circular dependency becomes easier to identify.

4 Evaluation

In this section, we report our evaluation of our tooling.

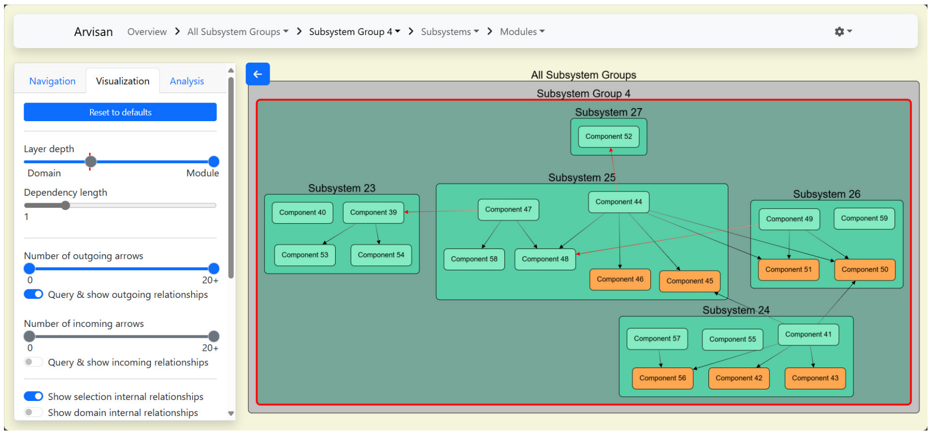

Fig. 3. The dependencies within **Subsystem Group 4** on the level of components. A circular dependency exists between **Subsystem 25** and **Subsystem 26**, because of an unintended dependency (red arrow) from **Component 49** to **Component 48**. The components colored in orange contain subsystem interfaces.

4.1 Study Design

We conducted a formative assessment of our adapted ARViSAN by employing it at Thermo Fisher Scientific on the TEM software. The goal of this evaluation was to identify the benefits of our solution in addressing the needs of the TEM architects and developers. Specifically, these needs included a clear overview of the TEM software architecture and support for monitoring the structural conformance between the reference and implemented architecture. Additionally, we aimed to collect challenges or unintended consequences of our tooling. For this purpose, we parsed and visualized the dependencies of the latest release of the system and conducted five 30-minute demonstrations with groups of company architects. In Table 1, we present an overview of the participants in these demonstrations (in order of conduct). Some of the architects were involved in subsystem architectures (S_1, S_2, S_4), and some in the overall TEM architecture (S_3). Additionally, we demonstrated our visualizations to two architects responsible for a separate Thermo Fisher Scientific system to diversify our set of participants (S_5).

We started each demonstration by showing the reference architecture diagram (cf. Figure 1) and the Graphviz graph created with the old tooling available at Thermo Fisher Scientific. Then, we presented the visualization we created in ARViSAN and allowed the architects to explore the dependencies most relevant to them. We asked the participants to report on their perceptions of ARViSAN and the visualizations we integrated, particularly compared to any tools they used for the same purpose so far. Each demonstration was led by the first author, who answered questions and documented the participants' feedback.

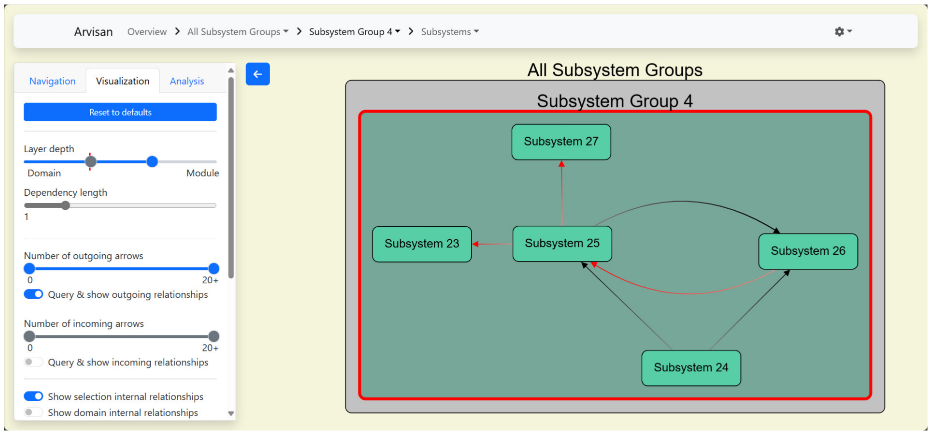

Fig. 4. The dependencies within **Subsystem Group 4** on the level of subsystems. Because dependencies are lifted to the subsystem level, the circular dependency between **Subsystem25** and **Subsystem26** is clearly visible—in contrast to Figure 3.

4.2 Results

Next, we present our observations from the demonstrations.

ARViSAN Compared to Other Tools Used. Our participants reported that they had access to various quality-related tools and dashboards. However, only a few of them reported using these tools to monitor architecture violations. Two participants (p_2, p_{12}) mentioned using tools for visualizing and verifying violations. Interestingly, p_2 relied on custom scripts, which were simplistic and required significant manual effort. In contrast, p_{12} had previously used NDepend, which is limited to software written in C# and .NET. They noted that they appreciated the flexibility and technology independence of our adapted ARViSAN; even though the required preprocessing of dependency data makes it less convenient. Lastly, p_3 used a search engine capable of cross-referencing source code. They pointed out that this engine lacked comprehensive dependency information, demanding extensive manual reviews.

Visualizing Violations. During the demonstrations, participants were able to identify unexpected and unintended dependencies. Specifically, in S_1, p_1 and p_2 noticed two unexpected dependencies to other subsystems. One was an unnecessary dependency. The cause for the second dependency remained unclear. In turn, p_1 indicated that being able to see dependencies is a positive initial step towards understanding and addressing architecture violations. During S_2, p_3 and p_4 quickly identified a circular dependency between two subsystems, which was distinctly visible. This dependency was already scheduled for refactoring, but the participants appreciated that our visualization made it immediately recognizable.

Visual Appeal. Several participants across the demonstrations appreciated the visual aspects of our tooling. For example, p_8 expressed that "*looking at a*

Table 1. Overview of our demo sessions and participants.

Session ID	Participant ID	Role	Assignment
S_1	p_1	Software Engineer	Subsystem49
	p_2	Domain Architect	
S_2	p_3	Domain Architect	Subsystem25
	p_4	Domain Architect	Subsystem26
S_3	p_5	Software Engineer	Entire system
	p_6	Staff Architect	
	p_7	Staff Architect	
	p_8	Senior Staff Architect	
	p_9	Senior Staff Architect	
	p_{10}	Senior Staff Architect	
S_4	p_{11}	Software Engineer	Subsystem7
	p_{12}	Domain Architect	
S_5	p_{13}	Senior Staff Architect	Other system
	p_{14}	Senior Staff Architect	

red edge immediately triggers you to start thinking about it and whether you can improve it." All participants found the tool intuitive and easy to use, particularly because it did not require learning complex query languages. Both p_1 and p_4 appreciated the ability to focus on specific nodes and to constrain the visualization to relevant dependencies. More generally, p_3 and p_5 emphasized the benefits of a visualization, with p_5 stating: "*The biggest added value to have a tool is to point it out [when people avoid following the architecture constraints], because it is hard to see, sometimes you find it by accident.*"

Scope and Depth. Given the layered architecture of the TEM system, several participants were interested in the scope and depth that can be visualized. Three participants (p_3, p_4, p_{12}) expressed a need for in-depth visualizations, though their preferences varied: from project-level details within a component to fine-grained code-level information, such as classes and functions. Additionally, p_3 noted that information on interface-level dependencies would be particularly useful, since they currently had to verify them manually—which is time-consuming due to a large number of interfaces. In contrast, p_5 and p_8 expressed concerns about expanding the visualization depth. For instance, p_5 explained: "*Going to code level makes the visualization too big, overloads you with too much data. This is a simple tool that shows where we are, and we can simply say 'fix these' [violations], we can have a different tool on a lower level, because optimizing language-specific dependencies is a different story.*" Participant p_8 stated they had a preference for other professional tools when it comes to the code level.

Suggested Use Cases. Participants in all demonstrations expressed enthusiasm for gaining access to our tooling and using it on their own. Several potential use cases emerged during our discussions, especially in S_2, S_3, and S_4.

In S_2 and S_3, participants were interested in using the tool to analyze dependencies across software releases. Although ARViSAN currently does not support direct visualizations and comparisons of multiple graphs, such analyses can be performed manually and can be automated in future work. During S_3, one participant suggested deploying the tool to development teams, so that it can be used in planning to improve the visibility and organization of refactoring tasks. In S_4, participants identified an opportunity to use the tool for onboarding new developers by providing a visual and interactive representation of the system architecture, which is in line with previous research [14, 15].

Limitations. Through the demonstrations, we identified limitations of our visualizations that we will tackle in the future. First, due to the diverse technologies used across the TEM system, dependencies are defined in different ways. As a result, some dependencies were missing from our visualizations. While a unification on how to declare dependencies is underway at Thermo Fisher Scientific, p_8 suggested that dependencies missing in the visualizations could also help identify where such unification is needed (i.e., an additional use case). Second, p_5 was concerned that when the entire TEM system is rendered in ARViSAN on the component level, violations can exist or may be introduced without being noticed (e.g., Figure 3 versus Figure 4). Currently, our solution can be used to interactively explore and display dependency violations. However, there is no functionality that prevents developers from introducing violations, which is also out of the scope of this work.

Tool Adoption. Adopting our tooling at Thermo Fisher Scientific was also a topic of interest among our participants. Previously, another open-source tool for architecture analysis was introduced within the company [2]. While the tool allowed to derive the implemented system architecture and facilitated large-scale refactoring, it required resource-intensive and time-intensive source code parsing. A dedicated team was assigned to maintain it, but due to high maintenance efforts and limited adoption, the tool was eventually discontinued.

Our solution does not depend on source code parsing and is meant to provide a visual and interactive architecture view. Still, because of their experiences with the previous tool, some participants were cautious about introducing another tool for architecture analysis. Currently, our extended version of ARViSAN and dependency processing scripts have been forked into an internal repository of Thermo Fisher Scientific, along with the parsed TEM system data as well as user and developer documentation. This enables developers and architects at Thermo Fisher Scientific to explore our tool within their workflows and for other systems.

5 Discussion

In the following, we discuss our results, distinguishing between implications for researchers, practitioners, and both.

5.1 Implications for Researchers

Improving Usability and Performance. When applied to large industrial software systems, dependency visualizations tend to become overly complex with hundreds of nodes and edges (e.g., the graph produced by Graphviz). We observed that while such a view contains rich information, participants found it overwhelming and unusable. Our participants valued the ability to simplify views by lifting dependencies to a higher level or by focusing on its relevant parts. Such functionalities and visually distinguishing violations (e.g., with different colors) made it easier to identify violations. We believe that:

> *The trade-offs between visual simplicity and depth of information should be further investigated to optimize the usability of software architecture visualization tools and to avoid overwhelming users.*

Studying Architecture Evolution. Several participants expressed interest in analyzing and visualizing subsystem dependencies across system releases. Recent research has explored this direction regarding the evolution of architecture smells and violations [5, 6, 24]. We believe that there is a need to also provide such evolution-based techniques for other properties of software architectures:

> *Researchers should investigate and compare existing techniques and solutions for studying the evolution of software architectures across releases to facilitate long-term software maintenance.*

5.2 Implications for Practitioners

Unifying Dependency Management Across Technologies. Implementing software that uses different programming languages and technologies can result in varying methods for declaring dependencies. Such differences hinder collecting and analyzing dependencies automatically, with some companies implementing their own techniques for declaring multi-lingual dependencies to mitigate such problems [7]. Declaring dependencies in a uniform way enables further automation, so that:

> *Companies may benefit from standardizing dependency-declaration practices or from applying tools that aggregate dependencies from multiple sources. This can facilitate the application of analysis and visualization tools, and can reduce the need for manual validation.*

Using Architecture Visualizations for Onboarding. Several participants saw value in using our visualizations for planning refactorings and onboarding new developers. This aligns with our prior work on developers' preferences for documentation [14, 15] and software architecture explanations [22]. However, we [22] also observed that architecture visualization tools are rarely used when onboarding new developers in practice. Therefore, we suggest:

> *Organizations should explore embedding architecture visualizations into the onboarding of new developers and in the planning of development work to facilitate better architectural decisions.*

5.3 Implications for Researchers and Practitioners

Validating Reference Against Implemented Architecture. We used a reference architecture as a baseline against which the implemented architecture can be validated. However, we observed that not all the latest developments of the TEM system were reflected in the reference architecture. For example, some documented subsystem groups were implemented as subsystems. Moreover, several implemented subsystems were not covered in the reference architecture. Therefore, we conclude that:

> *To improve architecture conformance, both the intended and the implemented architecture must be validated and revised if necessary. Future research should investigate how to synchronize the documentation and implementation of evolving architectures.*

Reducing Efforts of Architecture Data Preparation. To construct the reference architecture, we relied on documentation prepared by the software architects. Although this documentation provided valuable insights, certain architectural details, such as the mapping of components to subsystems, were not included. To fill these gaps, we conducted several meetings with the architects to collect and validate the missing information. We recognize that in other settings the effort required to obtain such architectural data may differ significantly. Based on our experience, we suggest the following recommendation to practitioners:

> *Assign clear ownership of architectural knowledge and documentation to support its maintenance and ongoing validation.*

For researchers, we argue:

> *Future research should examine not only the costs and efforts of repairing architecture conformance violations, but the efforts of gathering and maintaining architectural data as well as optimizing such efforts.*

Simplifying the Adoption of Tools. Our participants were enthusiastic about using our tool themselves. However, they raised concerns about the adoption due to experiences with previous tools. Specifically, some participants were concerned because a previously adopted open-source tool was discontinued due to high maintenance efforts that were coupled with limited use. Such concerns raise the question of what factors drive or hinder the adoption of (open-source) architecture tools in practice. For researchers, we argue that:

> *Future research should examine organizational and technical barriers to sustainably adopting and maintaining (open-source) tools in industrial settings.*

For practitioners, we suggest that:

> *Companies should assign clear ownership and support mechanisms for tool adoption. Integrating tools into existing workflows and developer environments, rather than providing stand-alone solutions, could improve tool adoption.*

6 Threats to Validity

We are aware of different threats to the validity of our work and discuss them according to the classification proposed by Runeson et al. [23] for case studies.

Construct Validity. Construct validity concerns the constructs we aimed to study. Since we investigated architecture conformance, a possible threat is that the violations our tool identified were incorrect. To mitigate this threat, we used a specification derived from the documented Thermo Fisher Scientific reference architecture to distinguish violations. We created and refined this specification with architects at Thermo Fisher Scientific.

Internal Validity. Internal validity concerns whether the observations in our study truly resulted from our proposed solution and not from other factors. One potential threat to the internal validity arises from the sample of participants who attended the demonstrations. They were invited by the second author as the company representative in our project. So, our participants may not fully represent the broader user population, since they are from one company, and we may have missed other interested stakeholders. To mitigate this threat, we invited experts from various roles and supplemented our demonstrations with a session (S_5) involving experts from a different Thermo Fisher Scientific system.

External Validity. Threats to the external validity concern the extent to which the study findings can be generalized. An inherent threat of industry reports is that, although our visualization tool is technology-independent, we designed our tooling with the specifics of Thermo Fisher Scientific in mind and evaluated it within the company only. However, a layered architecture may not be applicable or translate to paradigms like microservices. Additionally, using Thermo Fisher Scientific's reference architecture and tooling limits the generalizability if other companies do not have these resources—but using these is also a new idea of our work. Lastly, Thermo Fisher Scientific's context seems representative of (larger) industrial settings, but further studies are needed to validate the broader applicability of our tool in other organizations. Finally, a potential threat to the external validity is the lack of feedback on the long-term use of our tool. While the feedback from the demonstrations illustrates our tool's capabilities, we have not evaluated how ARViSAN performs or is perceived over an extended period. For this reason, the long-term impact and generalizability of our findings must be further verified in future work.

7 Conclusion

In this paper, we presented our experiences of extending and applying a visualization tool to support the needs of software architects by visualizing architectures and checking architecture conformance. We collaborated with experienced software architects and developers at Thermo Fisher Scientific and jointly evaluated the benefits and challenges of using our tool on the company's software systems. ARViSAN creates a hierarchical view of the software architecture, and visualizes violations of the intended architecture, such as unexpected and circular dependencies. The feedback from 14 software experts at Thermo Fisher Scientific indicated that the visualizations made it easier to focus on relevant system areas and to effectively detect architecture violations. These experts expressed great enthusiasm for using the tool in their daily work. Our insights suggest that existing software architecture visualization tools can aid software architects in maintaining conformance between the intended and the implemented software architectures. In the future, we will extend and improve our tooling, including its evaluation on other paradigms, other systems, and over time.

Data Availability Statement. We publish our anonymized dataset [28] and tooling [11, 12, 27] through persistent open-access repositories.

Acknowledgments. This work has received funding from Eindhoven University of Technology's IMPULSE program (COMARIC project, grant RVO-TKI2212P17).

Disclosure of Interests. Michel R. V. Chaudron and Filip Zamfirov have received funding from Eindhoven University of Technology and Thermo Fisher Scientific through the COMARIC project, which is funded from the Holland High Tech IMPULSE program. Andrei Radulescu works for Thermo Fisher Scientific.

References

1. Azadi, U., Fontana, F.A., Taibi, D.: Architectural smells detected by tools: A catalogue proposal. In: TechDebt (2019)
2. Dams, D., Mooij, A., Kramer, P., Rădulescu, A., Vaňhara, J.: Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code. In: SANER. IEEE (2018)
3. Fontana, F.A., Locatelli, F., Pigazzini, I., Mereghetti, P.: An architectural smell evaluation in an industrial context. In: ICSEA. IARIA (2020)
4. Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Di Nitto, E.: Arcan: A tool for architectural smells detection. In: ICSAW. IEEE (2017)
5. Gnoyke, P., Schulze, S., Krüger, J.: On developing and improving tools for architecture-smell tracking in Java systems. In: SCAM. IEEE (2023)
6. Gnoyke, P., Schulze, S., Krüger, J.: Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells. Journal of Systems and Software (2024)
7. Groot, T., Ochoa Venegas, L., Lazăr, B., Krüger, J.: A catalog of unintended software dependencies in multi-lingual systems at ASML. In: ICSE-SEIP. ACM (2024)
8. Hölzl, M., Rauschmayer, A., Wirsing, M.: Engineering of software-intensive systems: State of the art and research challenges. In: Software-Intensive Systems and New Computing Paradigms: Challenges and Visions. Springer (2008)

9. ISO/IEC/IEEE: Systems and software engineering–architecture description. Tech. Rep. ISO/IEC/IEEE 42010: 2011 (E)(Revision of ISO/IEC 42010: 2007 and IEEE Std 1471-2000), ISO/IEC/IEEE (2011)
10. Kakkenberg, R., Rukmono, S.A., Chaudron, M.R.V., Gerholt, W., Pinto, M., de Oliveira, C.R.: Arvisan: An interactive tool for visualisation and analysis of low-code architecture landscapes. In: MODELS. ACM (2024)
11. Kakkenberg, R., Zamfirov, F.: Software-analytics-visualisation-team/arvisan- backend: Arvisan backend v1.0.0 (Jul 2025). https://doi.org/10.5281/zenodo.15848517, https://doi.org/10.5281/zenodo.15848517
12. Kakkenberg, R., Zamfirov, F.: Software-analytics-visualisation-team/arvisan- frontend: Arvisan frontend v1.0.0 (Jul 2025). https://doi.org/10.5281/zenodo.15848538, https://doi.org/10.5281/zenodo.15848538
13. Kruchten, P.B.: The 4+ 1 view model of architecture. IEEE Software **12**(6), 42–50 (1995)
14. Krüger, J., Hebig, R.: What developers (care to) recall: An interview survey on smaller systems. In: ICSME. IEEE (2020)
15. Krüger, J., Hebig, R.: To memorize or to document: A survey of developers' views on knowledge availability. In: PROFES. Springer (2024)
16. Kumar, N.: Software architecture validation methods, tools support and case studies. In: ERCICA. Springer (2016)
17. Li, R., Liang, P., Soliman, M., Avgeriou, P.: Understanding software architecture erosion: A systematic mapping study. Journal of Software: Evolution and Process (2022)
18. Martin, R.: OO design quality metrics: An analysis of dependencies (1994)
19. Martini, A., Fontana, F.A., Biaggi, A., Roveda, R.: Identifying and prioritizing architectural debt through architectural smells: A case study in a large software company. In: ECSA. Springer (2018)
20. Mo, R., Snipes, W., Cai, Y., Ramaswamy, S., Kazman, R., Naedele, M.: Experiences applying automated architecture analysis tool suites. In: ASE (2018)
21. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes (1992)
22. Rukmono, S.A., Zamfirov, F., Ochoa, L., Chaudron, M.R.V.: From expert to novice: An empirical study on software architecture explanations (2025), https://arxiv.org/abs/2503.08628
23. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case study research in software engineering: Guidelines and examples. Wiley (2012)
24. Sas, D., Avgeriou, P., Fontana, F.A.: Investigating instability architectural smells evolution: An exploratory case study. In: ICSME. IEEE (2019)
25. Sas, D., Avgeriou, P., Uyumaz, U.: On the evolution and impact of architectural smells—An industrial case study. Empirical Software Engineering (2022)
26. Wan, Z., Zhang, Y., Xia, X., Jiang, Y., Lo, D.: Software architecture in practice: Challenges and opportunities. In: ESEC/FSE. pp. 1457–1469. ACM (2023)
27. Zamfirov, F.: Software-analytics-visualisation-team/arvisan- dependency-parser: Arvisan dependency parser v1.0.0 (Jul 2025). https://doi.org/10.5281/zenodo.15848528, https://doi.org/10.5281/zenodo.15848528
28. Zamfirov, F., Andrei, R., Krüger, J., Chaudron, M.: Software subsystem dependencies: Anonymized industrial dataset (Mar 2025). https://doi.org/10.5281/zenodo.15075337, https://doi.org/10.5281/zenodo.15075337